Semantics of Transient Connectors in Rewriting Logic

José Luiz Fiadeiro
Michel Wermelinger
José Meseguer

DI–FCUL TR–98–9

December 1998

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1700 Lisboa
Portugal

Technical reports are available at `http://www.di.fc.ul.pt/biblioteca/tech-reports`. The files are stored in PDF, with the report number as filename. Alternatively, reports are available by post from the above address.

Semantics of Transient Connectors in Rewriting Logic

José Luiz Fiadeiro
Departamento de Informática, Faculdade de Ciências
Universidade de Lisboa, Campo Grande, 1700 Lisboa, Portugal
E-mail: llf@di.fc.ul.pt

Michel Wermelinger
Departamento de Informática, Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa, 2825 Monte da Caparica, Portugal
E-mail: mw@di.fct.unl.pt

José Meseguer
Computer Science Laboratory, SRI International
333 Ravenswood Ave, Menlo Park, CA 94025, USA
E-mail: meseguer@csl.sri.com

Abstract

Research in Software Architectures has put forward the concept of connector to express complex relationships between system components, thus facilitating the separation of coordination from computation. A system can then be understood, at a given level of abstraction, in terms of its components and the connectors that establish how they interact. However, for systems in which many interconnections exist between their components, the architectures themselves may become very complex due to the high number of connectors in place. This is especially true in the context of mobile systems in which the interconnections are, by nature, transient in the sense that, at a given instant of time, only a subset of the potential connectors are actually effective. In this paper, we formalise a notion of transient connector that allows, at any given moment, for the architecture to depict only the connectors that are active and, in this way, capture the dynamics of architectures themselves. Our approach is based on the use of COMMUNITY, a UNITY-like program design language that has a semantics in Category Theory, and rewriting logic as a means of capturing the dynamic aspects of connectors.

Keywords: connectors, transient interactions, rewriting logic

1 Introduction

In a previous paper [19] we have argued in favour of a disciplined approach to mobility through the use of connectors (in the sense of software architectures). The idea is that mobility within a system can be characterised by the transient nature of the interconnections that exist between the components of the system. Because, from an architectural point of view, such interconnections are best captured through the use of

connectors [18], changes in the interconnections should be also captured at the level of the connectors that are in place.

For that purpose, we defined connectors which, through guarded actions, were able to set or reset interconnections between components according to given conditions of applicability (coded in the guards of the actions). However, because the dynamic behaviour of the system may require a considerable number of different situations in which such interconnections should apply, the architecture of the system may get cluttered by a high number of connectors, even if, at each given time, only a few of the applicability conditions hold.

To circumvent this problem, we suggested in [20] the concept of a transient connector in the sense of a connector with an associated condition on the state of its roles that determines the situations in which it applies. The idea is that a connector does not need to be permanently part of an architecture, but is added and removed according to its applicability condition. This can be seen as a restricted form of dynamic architectures in which the evolution of the architecture is determined by well defined operations of addition and removal of connectors that are to be performed in well determined states of the underlying system.

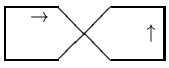
Our purpose in this paper is to expand the original motivation and further develop the notion of transient connector in the context of dynamic architectures, namely by providing a well-defined mathematical semantics through which the evolution of the architecture can be inferred and reasoned about. Capitalising on previous work on the formalisation of architectural connectors in general [6], and connectors for mobile systems in particular [19], we use Category Theory to represent software architectures. For modelling the dynamic aspects of architectures, we use Rewriting Logic [11], a formalism that has already been applied to the formalisation of several architectural aspects of systems, e.g. [12, 13]. We illustrate the approach with a connector for partial synchronisation of actions written in COMMUNITY [7].

2 Preliminaries

In this section we present the basic notions of Category Theory and COMMUNITY. Due to space limitations, we focus only on those aspects essential to understand the rest of the paper and omit all technical details. This section also introduces the example application.

2.1 The Example

Our example is inspired in the luggage distribution system also used to illustrate Mobile UNITY [17]. One or more carts move on a N units long track with the shape



A cart advances one unit at each step in the direction shown by the arrows. The i-th cart starts from a unit determined by an injective function *start* of i. Carts are continuously moving around the circuit. Their movement must be synchronised in such a way that no collisions occur at the crossing.

2.2 Category Theory

Category Theory [15] is the mathematical discipline that studies, in a general and abstract way, relationships between arbitrary entities. A category is a collection of objects together with a collection of morphisms between pairs of objects. A morphism f with source object a and target object b is written $f : a \to b$. Morphisms come equipped with a composition operator ";" such that if $f : a \to b$ and $g : b \to c$ then $f;g : a \to c$. Composition is associative and has identities id_a for every object a.

Diagrams are directed graphs—where nodes denote objects and arcs represent morphisms—and can be used to represent "complex" objects as configurations of smaller ones. For categories that are well behaved, each configuration denotes an object that can be retrieved through an operation on the diagram called colimit. Informally, the colimit of a diagram is the "minimal" object such that there is a morphism from every object in the diagram to it (i.e., that contains the objects in the diagram as components) and the addition of these morphisms to the original configuration results in a commutative diagram (i.e., interconnections, as established by the morphisms of the configuration diagram, are enforced).

2.3 Community

COMMUNITY [7] is a program design language based on UNITY [2] and IP [8]. In this paper we only consider a subset of the full language. For our purposes, a COMMUNITY program consists of a set of typed attributes, a boolean expression to be satisfied by the initial values of the attributes, and a set of actions, each of the form *name*: [*guard* $\to$ *assignment(s)*]. The empty set of assignments is denoted by `skip`. Action names act as *rendez-vous* points for program synchronisation. At each step, one of the actions is selected and, if its guard is true, its assignments are executed simultaneously. To be more precise, syntactically a program has the form

```
program P is
var     V
read    R
init    I
do      a1: [g1 →  v11 := exp11 || v21 := exp21 || ...]
[]      a2: [g2 → v12 := exp12 || ...]
[]      ...
```

where R are external attributes (i.e., the program may not change their values), V are the local attributes, I is the initialisation condition on V, a_i are the actions with boolean expressions g_i over $V \cup R$, $v_{ij} \in V$, and exp_{ij} expressions over $V \cup R$.

The following program describes the behaviour of the i-th cart, as introduced in section 2.1.

```
program Cart_i is
var     l : int
init    l = start(i)
do      move: [true → l := (l + 1) mod N]
```

A morphism from a program P to a program P' states that P is a component of the system P' and, as shown in [7], captures the notion of program superposition [2, 8]. Mathematically speaking, the morphism maps each attribute of P into a attribute of P' of the same type, and it maps each action name g of P into a (possible empty) set of

action names $\{g'_1, \ldots, g'_n\}$ of P' [19]. Those actions correspond to the different possible behaviours of g within the system P'. These different behaviours usually result from synchronisations between g and other actions of other components of P'. Thus each action g'_i must preserve the functionality of g, possibly adding more things specific to other components of P'. In particular, the guard of g'_i must not be weaker than the guard of g, and the assignments of g must be contained in g'_i.

Putting it in a succinct way, a morphism $f : P \to P'$ maps the vocabulary of P into the vocabulary of P', and thus any expression e over the variables of P can be automatically translated into an expression $f(e)$ over the variables of P'. Thus if action a: $[g \to v := exp]$ of P is mapped into action a' of P' (possibly among others), then it must be the case that a': $[g' \to f(v) := f(exp) \parallel \ldots]$ with g' implying $f(g)$.

Continuing the example, the following diagram shows in which way program "Cart$_i$" is a component of a program that also counts how many laps have been completed by checking how often the cart passes by a position (e.g., the crossing) given by the environment.

```
program Cart_i is
var     l : int
init    l = start(i)
do      move: [true → l := (l + 1) mod N]
```

$move \mapsto \{move, pass\}$ ↓ $l \mapsto location$

```
program CartWithLaps_i is
var     location, laps : int
read    position : int
init    location = start(i) ∧ laps = 0
do      pass: [location = position
        → location := (location + 1) mod N || laps := laps + 1]
[]      move: [location ≠ position → location := (location + 1) mod N]
```

Notice how the second program strengthens the initialization condition and it divides action "move" in two sub-cases, each satisfying the condition given above. We henceforth omit the "mod N" operation and the action guards whenever they are "true".

It can be proved that COMMUNITY programs and their morphisms form a category in which every finite diagram has a colimit, which, by definition, is the minimal program that contains all programs in the diagram. Thus the diagram specifies the architecture and the colimit represents the resulting system. Since the proof of the existence of a colimit is constructive, the architecture can be "compiled" into a single program that simulates the execution of the overall system.

3 Permanent Connectors

We first briefly recapitulate our approach to connectors for mobile programs [19] because they form the semantic basis of transient connectors, to be presented in the next section.

A n-ary connector consists of n roles R_i and one glue G stating the interaction between the roles. These act as "formal parameters", restricting which components may be linked together through the connector. Thus, the roles may contain attributes and actions which are not used for the interaction specification.

In Category Theory, all relationships between objects must be made explicit through

morphisms. In the particular case of COMMUNITY programs, it means for example that two attributes (or actions) of two unrelated programs are different, even if they have the same name. To state that attribute (or action) a_1 of program P_1 is the same as attribute (resp. action) a_2 of P_2, even if a_1 and a_2 have the same names, one needs a third, "mediating" program C—the channel—containing just an attribute (resp. action) a and two morphisms $\sigma_i : C \to P_i$ that map a to a_i. Stating that two actions are the same means synchronising them. In general, a channel contains the features that are shared between the two programs it is linked to, thus establishing a symmetrical and partial relationship between the vocabularies of those programs.

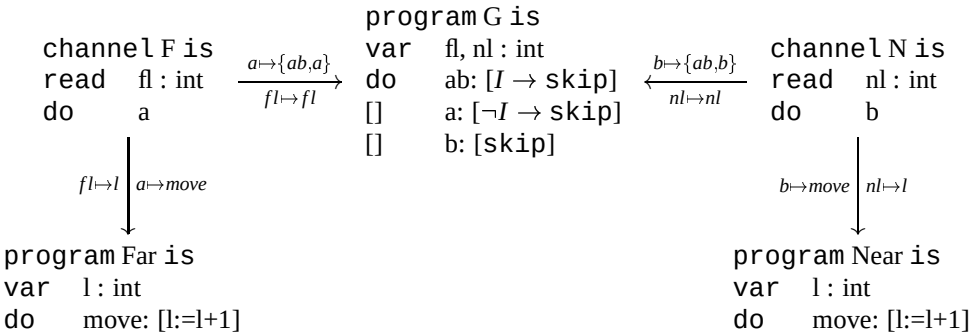
Applying these ideas to connectors [6], for each role R_i there must be a channel C_i together with morphisms $\gamma_i : C_i \to G$ and $\rho_i : C_i \to R_i$ stating which attributes and actions of R_i are used in the interaction specification, i.e., the glue. As channels just establish the required relationships between action names, their actions are always of the form a: [true → `skip`], and thus in this paper we use the abbreviated notation

`channel` C `is`
`read` $v_1 : T_1; \ldots$
`do` $a_1, a_2, \ldots$

The categorical framework also allows one to make precise when an n-ary connector can be applied to components $P_1, \ldots, P_n$, namely when morphisms $\iota_i : R_i \to P_i$ exist. This corresponds to the intuition that the "actual arguments" (i.e., the components) must instantiate the "formal parameters" (i.e., the roles). As an illustration, an instantiated binary connector has the diagram

$$P_1 \xleftarrow{\iota_1} R_1 \xleftarrow{\rho_1} C_1 \xrightarrow{\gamma_1} G \xleftarrow{\gamma_2} C_2 \xrightarrow{\rho_2} R_2 \xrightarrow{\iota_2} P_2$$

Returning to our example, assume that two carts are approaching the crossing and one of them is nearer to it. To avoid a collision it is sufficient to force the nearest cart to move whenever the most distant one does. That can be achieved using an action subsumption connector. Action a subsumes action b if b executes whenever a does. This can be seen as a partial synchronisation mechanism: a is synchronised with b, but b can still execute freely. The connector that establishes this form of interaction is



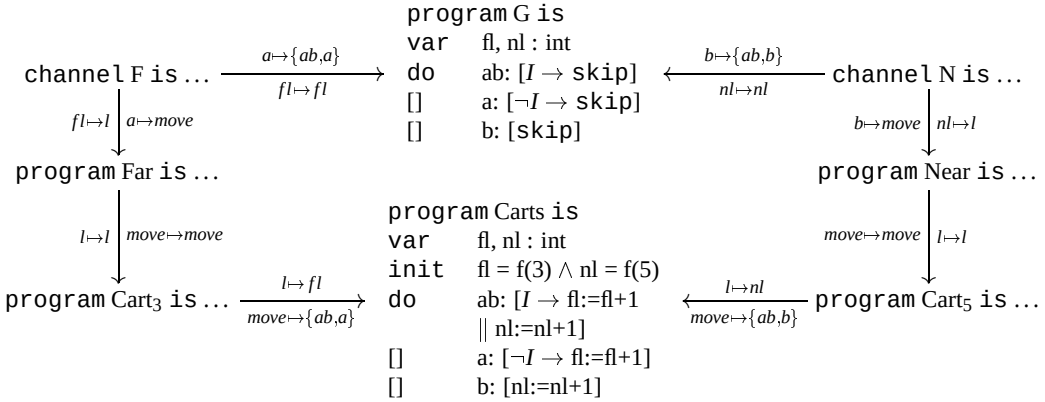
where I is the interaction condition. Notice that although the two roles are isomorphic, the binary connector is not symmetric because the glue treats the two actions differently. This is clearly indicated in the glue: "b" may be executed alone at any time, while "a" must co-occur with "b" if the interaction is taking place. Hence, action "a" is the one that we want to connect to the "move" action of the cart that is further away from the crossing, while action "b" is associated to the movement of the nearest cart (the one that will instantiate role "Near").

To complete the example, it remains to show what the interaction condition is, and what system is obtained through role instantiation. Assuming that track units 7 and 28 cross and that movement coordination should start when both carts are at most 3 units away from the crossing, one has

$$I = 0 \leq 7 - nl < 28 - fl \leq 3 \vee 0 \leq 28 - nl < 7 - fl \leq 3$$

The first disjunct treats the case when the nearest cart is moving towards track unit 7 and the other cart is approaching unit 28. The other disjunct handles the opposite case.

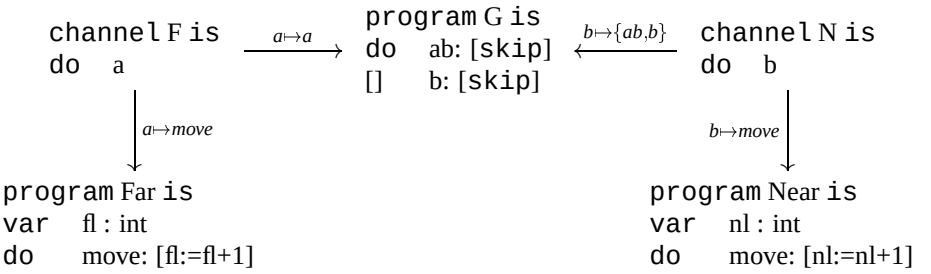
The roles omit the initialisation condition of the location attribute, so that they can be instantiated with any particular cart. The next diagram shows the application of the connector to carts 3 and 5, assuming the latter is nearer to the crossing, and the resulting colimit.



4 Transient Connectors

The previous approach has two drawbacks. First, the specification may grow quite large, because the architecture of the system is given by a fixed diagram showing *all* possible connections between the existing components, even if due to the actual computations some components will never interact. In our example the diagram for an architecture with c carts has $c^2 - c$ copies of the action subsumption connector, one for each pair $\langle Cart_i, Cart_j \rangle$ with $i \neq j$. Altogether, the specification is a graph with c^2 nodes and $2 \times (c^2 - c)$ arcs. The second disadvantage is that because the interaction condition is coded into the guards of the actions of the glue in order to show explicitly when they can be executed, it can only be changed if one has access to the implementation of the glue. In other words, it is not possible to provide libraries of pre-compiled connectors to be (re)used for many applications or in different situations.

To obviate these problems we proposed in [20] the use of transient connectors as consisting of a pair $\langle I, T \rangle$ where T is a connector and I is a boolean expression, called interaction condition, over the attributes of its roles. Returning to our example, the transient action subsumption connector is composed of the connector



and the same interaction condition as in the previous section.

It is important to compare the two solutions to appreciate what is being gained. The most obvious difference is that the interaction condition was moved outside the connector, more precisely, outside the glue of the connector. This move triggers some simplifications. With permanent connectors, the glue, the channels, and their morphisms had to contain all the relevant attributes from the roles because the interaction condition was inside the glue. This is no longer necessary with transient connectors because the condition is stated directly in terms of the attributes of the roles. This means that, apart from making sure that attributes in different roles have different names, writing connectors becomes simpler (and thus less error-prone).

However, whereas with permanent connectors the configuration of a system could be represented in a mathematically simple way as a (complex) diagram whose colimit returns the behaviour of the system, with transient connectors we avoid the explosion of connectors that clutters the configuration diagram, but we need to provide a new mathematical semantics for the notion of architecture. More precisely, we need a new semantics for the notion of connector and a new representation for the systems that are built from them.

The semantics that we outlined in [20] identifies the configuration of a system with two sets, one of transient connectors, the other of components. At any point in time, the diagram is constructed as follows. For every n-ary transient connector $\langle I, T\rangle$ and for every n components P_i, if there are morphisms from T's roles to the components such that the values of the attributes of the P_i corresponding to the attributes of the roles make I true, then the components P_i are connected by a new copy of T.

The semantics that we wish to present instead captures the dynamic flavour of the configurations in a more explicit way by formalising transient connectors as conditional rewrite rules over extended diagrams that represent anchored configurations. By an anchored configuration we mean a diagram in the category of COMMUNITY programs enriched, for each node, with a valuation for the attributes of the program that labels the node. Such valuations have to be compatible with the morphisms in the sense that, if we have an edge between nodes $\langle P, v\rangle$ and $\langle P', v'\rangle$ labelled by a morphism f between P and P', then every attribute a of P must be such that $v(a) = v'(f(a))$.

This approach has several advantages over the set-based representation. From a mathematical point of view, it identifies the configuration of a system with a diagram as was the case with permanent connectors; the difference now is that this diagram evolves as the system evolves. The advantage here is that we can perform the same kind of mathematical reasoning over configurations as before, like computing the colimit to obtain the global program that rules the current execution of the system. Again, the difference now is that this program may change as it executes.

Another advantage of this representation is that we can resort to well established approaches for rewriting diagrams in order to formalise transient connectors. The use

of graph rewriting in Software Architectures is not new, see for instance [14] for the use of graph grammars in the formalisation of architectural styles. However, we have chosen instead rewriting logic [11], a framework that still allows us to rewrite diagrams, but which has the advantage of accommodating other forms of rewriting structures. In particular, it will allow us to provide a more integrated semantics of the dynamics of configurations by combining both the evolution of the architecture and of the current state of the system. Rewriting Logic also has the advantage of making available an interpreter—Maude [3]—for the execution of the rewrite rules, thus providing us with an operational semantics for both the architectural connectors and the system actions. Moreover, several formal techniques are being defined for reasoning about the behavioural properties of rewriting theories that will enable us to analyse the evolution of the architectures themselves.

In the next section, we detail this semantics.

5 Transient connectors as rewrite rules

Rewriting logic [9, 10] expresses an essential equivalence between logic and computation in a particularly simple way. Namely, system *states* are in bijective correspondence with *formulas* (modulo whatever structural axioms are satisfied by such formulas: for example, modulo the associativity and commutativity of a certain connective) and concurrent *computations* in a system are in bijective correspondence with *proofs* (modulo appropriate notions of equivalence between computations and between proofs). Given this equivalence between computation and logic, a rewriting logic axiom of the form

$$t \longrightarrow t' \textbf{ if } C$$

has two readings. Computationally, it means that a fragment of a system's state that is an instance of the pattern t can *change* to the corresponding instance of t' concurrently with any other state changes when condition C holds; that is, the computational reading is that of a *local concurrent transition*. Logically, it just means that we can derive the formula t' from the formula t when C holds; that is, the logical reading is that of an *inference rule*.

Rewriting logic is entirely neutral about the structure and properties of the formulas/states t. They are entirely *user-definable* as an algebraic data type satisfying certain equational axioms, so that rewriting deduction takes place *modulo* such axioms. More precisely, a *signature* in rewriting logic is an equational theory (Σ, E), where Σ is an equational signature and E is a set of Σ-equations. Rewriting will operate on equivalence classes of terms modulo E. In this way, rewriting is made free from the syntactic constraints of a term representation and gain a much greater flexibility in deciding what counts as a *data structure*; for example, string rewriting is obtained by imposing an associativity axiom, and multiset rewriting by imposing associativity and commutativity. Of course, standard term rewriting is obtained as the particular case in which the set of equations E is empty. Techniques for rewriting modulo equations have been studied extensively [5] and can be used to implement rewriting modulo many equational theories of interest.

Given a signature (Σ, E), *sentences* of rewriting logic are sequents of the form

$$r : [t]_E \longrightarrow [t']_E \textbf{ if } C,$$

where r is a label, t and t' are Σ-terms possibly involving some variables, $[t]_E$ denotes the equivalence class of the term t modulo the equations E, and C is a condition ex-

pressed as a conjunction of equations or sequents of the form $[u_i] \longrightarrow [v_i]$. A *rewrite theory* R is a 4-tuple $R = (\Sigma, E, L, R)$ where Σ is a ranked alphabet of function symbols, E is a set of Σ-equations, L is a set of *labels*, and R is a set of sentences as described above, called *rewrite rules*.

Because of its neutrality with regard to the structure and properties of states and formulas, rewriting logic has good properties as a *semantic framework* [11], in which many different system styles and models of concurrent computation and many different languages can be naturally expressed without any distorting encodings.

For instance, the computational model of the parallel program design language UNITY [2] is easily expressed in rewriting logic. The details are given in [10], but the basic idea is straightforward. In essence, a UNITY program, and a COMMUNITY program for that matter, is a set of multiple assignment statements of the form

$$a_1, \ldots, a_n := exp_1(a_1, \ldots, a_n), \ldots, exp_n(a_1, \ldots, a_n)$$

where the a_i are declared attributes, and the $exp_i(a_1, \ldots, a_n)$ are Σ-terms for Σ a fixed many-sorted signature defined on the types of the declared attributes. A COMMUNITY program can be represented as a rewrite theory whose signature defines state configurations as sets of pairs $\langle a : T \mid val : v \rangle$ with a a program attribute, T a type, and v a value of type T, and every action a as a rewrite rule:

$$a: \begin{array}{l} \langle a_1 : T_1 \mid val : x_1 \rangle \ldots \langle a_n : T_n \mid val : x_n \rangle \\ \longrightarrow \langle a_1 : T_1 \mid val : exp_1(x_1, \ldots, x_n) \rangle \ldots \langle a_n : T_n \mid val : exp_n(x_1, \ldots, x_n) \rangle \end{array} \textbf{ if } g$$

where g, the guard of the action, is a condition on the x_i. In this case, the equational axioms E modulo which we rewrite are the associativity and commutativity of set union, which is expressed in such rule by empty syntax (juxtaposition).

As an example, consider the program Cart_i. The action *move* can be represented by

$$move : \langle l : int \mid val : x \rangle \longrightarrow \langle l : int \mid val : x + 1 \rangle$$

Graph rewriting has also been represented in rewriting logic [11]. Labelled graphs are axiomatised equationally as an algebraic data type in such a way that graph rewriting becomes rewriting modulo the equations axiomatising the type. Axiomatisations in this spirit include those of Bauderon and Courcelle [1], Corradini and Montanari [4], and Raoult and Voisin [16]. We adopt the axiomatisation given in [11], in which a (labelled) graph is viewed as a set of nodes, and thus graph rewriting is viewed modulo the associativity and commutativity of set union, expressed again with empty syntax. Each node is an object of the form

$$\langle identifier \mid label : L; links : K \rangle$$

containing label information and a list of edges, each of which consists of a pair

$$\langle identifier \mid target : N; label : L \rangle$$

identifying the target node and the label of the edge. For instance, the graph

$$\begin{array}{ccccc} C_1 & \xrightarrow{\gamma_1} & G & \xleftarrow{\gamma_2} & C_2 \\ \downarrow{\scriptstyle \delta_1} & & & & \downarrow{\scriptstyle \delta_2} \\ P_1 & & & & P_2 \end{array}$$

can be represented by the term

$\langle a_1 \mid label : C_1; links : \langle f_1 \mid target : c_1; label : \delta_1\rangle, \langle g_1 \mid target : b; label : \gamma_1\rangle\rangle$
$\langle a_2 \mid label : C_2; links : \langle f_2 \mid target : c_2; label : \delta_2\rangle, \langle g_2 \mid target : b; label : \gamma_2\rangle\rangle$
$\langle b \mid label : G; links : nil\rangle$
$\langle c_1 \mid label : P_1; links : nil\rangle$
$\langle c_2 \mid label : P_2; links : nil\rangle$.

The labels that interest us for the semantics of transient connectors are pairs (P,s) with P a COMMUNITY program and s a state configuration for P. Edges between nodes labelled (P',s') are labelled with morphisms $f : P \to P'$ such that, for every $\langle a : T \mid val : v\rangle$ in s, $\langle f(a) : T \mid val : f(v)\rangle$ is in s' (modulo the equational axioms E), i.e., morphisms have to respect the state configurations. We shall call such graphs *anchored configurations*.

The idea is to represent an n-ary transient connector defined by $\gamma_i : C_i \to G$ and $\rho_i : C_i \to R_i$ as a conditional graph rewrite rule of the form

$$
\begin{array}{lccccc}
 & C_1 & \xrightarrow{\gamma_1} & (G,s) & \xleftarrow{\gamma_n} & C_n \\
(P_1,s_1)\cdots(P_n,s_n) \longrightarrow & \downarrow{\scriptstyle \rho_1;X\iota_1} & & \cdots & & \downarrow{\scriptstyle \rho_n;X\iota_n} \\
 & (XP_1,s_1) & & & & (XP_n,s_n)
\end{array}
$$

if $I \wedge X\iota_1 \in morph(R_1,XP_1) \wedge \cdots \wedge X\iota_n \in morph(R_n,XP_n)$

where the XP_i are "variables" that can be instantiated with any programs subject to the conditions imposed by the rule, which are, for each instance P_i, that it admits the corresponding instance of s_i as a valid configuration, and that an instance of $X\iota_i$ be found that is a morphism from the connector's role R_i to the instance P_i (thus making P_i a true instance of the role in the categorical sense as discussed in section 3). Notice that each instance P_i will be connected to the glue via the channel C_i and the morphism that results from the composition of the morphisms that connect the channel to the role, as given by the connector, and the instance of $X\iota_i$ that establishes P_i as an instance of R_i. The instances of the state configuration must, of course, satisfy the interaction condition I. Finally, s is the state given by the initialisation condition of the glue G.

The corresponding rule in rewriting logic is obtained by taking the representation of the graphs:

$\langle XN_1 \mid label : (XP_1,s_1); links : XL_1\rangle \cdots \langle XN_n \mid label : (XP_n,s_n); links : XL_n\rangle$
$\longrightarrow$
$\langle a_1 \mid label : (C_1,nil); links : \langle f_1 \mid target : XN_1; label : \rho_1;X\iota_1\rangle\langle g_1 \mid target : b; label : \gamma_1\rangle\rangle$
$\cdots$
$\langle a_n \mid label : (C_n,nil); links : \langle f_n \mid target : XN_n; label : \rho_n;X\iota_n\rangle\langle g_n \mid target : b; label : \gamma_n\rangle\rangle$
$\langle b \mid label : (G,s); links : nil\rangle$
$\langle XN_1 \mid label : (XP_1,s_1); links : XL_1\rangle \cdots \langle XN_n \mid label : (XP_n,s_n); links : XL_n\rangle$
if
$I \wedge X\iota_1 \in morph(R_1,XP_1) \wedge \cdots \wedge X\iota_n \in morph(R_n,XP_n)$

where $XN_1,\ldots,XN_n$ are variables ranging over node identifiers, $XP_1,\ldots,XP_n$ are variables ranging over programs, $X\iota_1,\ldots,X\iota_n$ are variables ranging over morphisms, and $XL_1,\ldots,XL_n$ are variables ranging over lists of edges, and the identifiers a_1, ..., a_n, f_1, ..., f_n, and b are "new".

For instance, in the case of the cart synchronisation, we would have for the connector defined in section 4 the rewrite rule corresponding to:

$$(XP_1, \langle Xf : int \mid val : f\rangle, XL)(XP_2, \langle Xn : int \mid val : n\rangle, XM)$$
$$\longrightarrow$$

$$\begin{array}{ccccc} & & F & \xrightarrow{a\mapsto a} & (G, nil) & \xleftarrow{b\mapsto\{ab,b\}} & N \\ & & \Big\downarrow{\scriptstyle a\mapsto move;X\iota_1} & & & & \Big\downarrow{\scriptstyle b\mapsto move;X\iota_2} \\ (XP_1, \langle Xf : int \mid val : f\rangle, XL) & & & & & & (XP_2, \langle Xn : int \mid val : n\rangle, XM) \end{array}$$

if

$0 \leq 7 - n < 28 - f \leq 3 \vee 0 \leq 28 - n < 7 - f \leq 3 \wedge X\iota_1 \in morph(Far, P_1) \wedge X\iota_2 \in morph(Near, P_2)$

The expression of the rule in rewriting logic, through the use of variables ranging over nodes, programs, morphisms and lists of edges, makes clear that the left hand-side of the rules can be instantiated by any nodes labelled by any programs matching the given state configuration, and with any connectivity to other nodes, subject to the applicability conditions. These include the interaction condition of the programs and the identification of the morphisms that are being used to instantiate the roles. These conditions on the instantiation morphisms are essential to narrow down the scope of the applicability of the rule to programs that actually fit the roles.

Because the left-hand side of the rule is copied to the right hand side, the effect of the application of the rule is to superpose the glue and its connections to the components identified through the left-hand side. The fact that the identifiers of the superposed nodes and edges are new means that the interconnections are, indeed, new and do not interfere with other interconnections that may exist.

As an example of applying the rule, assume there are three carts freely moving around (i.e., there are no connectors) and their current positions are 30, 26 and 4. The corresponding configuration is given by

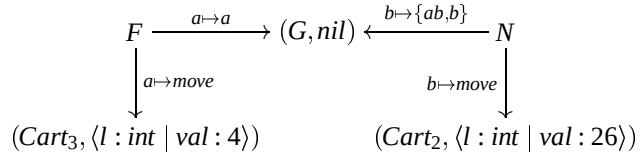
$$(\text{Cart}_1, \langle l : int \mid val : 30\rangle)\ (\text{Cart}_2, \langle l : int \mid val : 26\rangle)\ (\text{Cart}_3, \langle l : int \mid val : 4\rangle)$$

which is represented by the term

$\langle c_1 \mid label : (Cart_1, \langle l : int \mid val : 30\rangle); links : nil\rangle$
$\langle c_2 \mid label : (Cart_2, \langle l : int \mid val : 26\rangle); links : nil\rangle$
$\langle c_3 \mid label : (Cart_3, \langle l : int \mid val : 4\rangle); links : nil\rangle.$

The interaction is true if XP_1 is instantiated with Cart_3 (and Xf with l and f with 4) and XP_2 with Cart_2 (and thus Xn with l and n with 26). Since both the roles "Near" and "Far" (as well as the carts) have a single attribute and a single action, the only possible instantiation morphisms are: $\iota_1 = \{fl \mapsto l, move \mapsto move\}$ and $\iota_2 = \{nl \mapsto l, move \mapsto move\}$. Composing $\rho_1 = \{a \mapsto move\}$ with ι_1 one obtains $a \mapsto move$, and similarly for ρ_2 and ι_2. The applicability conditions are thus met and the new configuration is

$(\text{Cart}_1, \langle l : int \mid val : 30\rangle)$



whose representation is given by the term

$\langle c_1 \mid label : (Cart_1, \langle l : int \mid val : 30\rangle); links : nil\rangle$
$\langle c_2 \mid label : (Cart_2, \langle l : int \mid val : 26\rangle); links : nil\rangle$
$\langle c_3 \mid label : (Cart_3, \langle l : int \mid val : 4\rangle); links : nil\rangle$
$\langle a_1 \mid label : F; links : \langle f_1 \mid target : c_1; label : a \mapsto move\rangle, \langle g_1 \mid target : b; label : a \mapsto a\rangle\rangle$
$\langle a_2 \mid label : N; links : \langle f_2 \mid target : c_2; label : b \mapsto move\rangle, \langle g_2 \mid target : b; label : b \mapsto \{ab, b\}\rangle\rangle$
$\langle b \mid label : G; links : nil\rangle$.

Notice that the reverse rewrite rules, removing the application of the connectors, are also necessary when the interaction condition becomes false.

Summarising, the architecture of the system consists of a rewrite theory presentation over the signature that we have outlined above in terms of anchored configurations. The axioms of this rewrite theory presentation are the conditional rewrite rules defined by the connectors. Given an initial anchored configuration of the system, such a rewrite theory presentation provides us with the space of possible evolutions of the system configuration from that state.

The representation for COMMUNITY actions given above can be extended to anchored configurations by having the rewrite rules that represent actions to change only the state component, leaving the configuration unchanged. For instance, the representation of the *move* action used above for illustration, would be given by

$$move: \begin{array}{l} \langle XN \mid label : (Cart, \langle l : int \mid val : x\rangle); links : XL\rangle \\ \longrightarrow \langle XN \mid label : (Cart, \langle l : int \mid val : x+1\rangle); links : XL\rangle \end{array}$$

When we combine the rewriting of the system configuration in terms of connectors with the rewriting of its states in terms of its actions, it is important to define a discipline of execution that makes sure that the configuration is rewritten before the actions are given a chance of rewriting the state. This is because rewriting the structural configuration may change the way components are interconnected, and in particular the way actions are synchronised and, hence, it is necessary to compute which synchronisation sets of actions can occur only after the topology of the system has stabilised. This principle can be given an operational semantics in languages such as Maude [3]—that supports rewriting strategies through its reflective mechanisms—by assigning an eager evaluation discipline to the rewrite rules defined by the architectural connectors.

Because the rewrite rules defined by the connectors do not change the state of the components, their execution can be defined in a single transaction. That is to say, the applicability conditions of the connectors can all be evaluated on the same state and, hence, the set of applicable connectors can be determined over that same state and executed in any order, or concurrently if that is possible.

Finally, it is important to point out that, with the proposed semantics, it is fairly easy to change the architecture of a system just by changing the connectors that define the theory presentation. For instance, one can replace the subsumption connector by an action inhibition connector—making the cart that is further away to stop while the

one that is nearer crosses the intersection—just by changing the glue of the connector.

6 Concluding Remarks

Transient connectors state explicitly the condition that programs must obey in order to interact according to the way prescribed by the connector. Externalising the interaction condition makes the connectors simpler and allows their reuse under different circumstances. The architectural diagram also becomes simpler (and more intuitive) since it reflects at each point in time just the connections that are in place.

In this paper we have given an operational semantics for transient connectors in rewriting logic. For each connector there are two rules, one to introduce it into the architecture, the other one to remove it. In both cases, the left and right hand sides of the rules are anchored configurations showing the current state of each program, thus allowing the evaluation of the interaction condition of the connector. Programs are written in a UNITY-like language, which also has a semantics in rewriting logic. This allows a uniform representation of both the computational and the architectural levels, showing how they interact, and of their dynamics, showing how they jointly evolve.

There is also an added expressive power in the proposed semantics of architecture that we intend to explore in future work: the ability of actions to be constrained by conditions on the structure of the configuration. Further work that we intend to pursue includes the definition of rewriting strategies for execution in Maude, and the use of logical mechanisms available for reasoning about rewrite theories for reasoning about the evolution of the systems that are subject to transient connectors.

Acknowledgements

This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) through contracts PRAXIS XXI PCEX/P/MAT/46/96 (ACL) and PRAXIS XXI 2/2.1/TIT/1662/95 (SARA). Support from the US Office of Naval Research under contract N00014-96-C-0114, and the National Science Foundation through grant CCR-9633363 are also gratefully acknowledged. This work was produced while José Fiadeiro was on leave at the SRI International with the support of FLAD and FCT (contract BPD 16367/98).

References

[1] M. Bauderon and B. Courcelle. Graph expressions and graph rewriting. *Math. Systems Theory*, 20:83–127, 1987.

[2] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[3] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In *Proceedings of the First International Workshop on Rewriting Logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 65–89. Elsevier, 1996.

[4] A. Corradini and U. Montanari. An algebra of graphs and graph rewriting. In D. P. et al., editor, *Category Theory and Computer Science*, pages 236–260. Springer LNCS 530, 1991.

[5] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 243–320. North-Holland, 1990.

[6] J. L. Fiadeiro and A. Lopes. Semantics of architectural connectors. In *Proceedings of TAPSOFT'97*, volume 1214 of *LNCS*, pages 505–519. Springer-Verlag, 1997.

[7] J. L. Fiadeiro and T. Maibaum. Categorial semantics of parallel program design. *Science of Computer Programming*, 28:111–138, 1997.

[8] N. Francez and I. Forman. *Interacting Processes*. Addison-Wesley, 1996.

[9] J. Meseguer. Rewriting as a unified model of concurrency. In *Proceedings of the Concur'90 Conference, Amsterdam, August 1990*, pages 384–400. Springer LNCS 458, 1990.

[10] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[11] J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In *Proceedings of the 7th International Conference on Concurrency Theory*, volume 1119 of *LNCS*, pages 331–372. Springer-Verlag, 1996.

[12] J. Meseguer and C. Talcott. Semantic interoperation of dynamic heterogeneous architectures, 1997. Technical presentations to EDCS Architecture Cluster Meeting, April and July 1997.

[13] J. Meseguer and C. Talcott. Using rewriting logic to interoperate architectural description languages (I and II). Lectures at the Santa Fe and Seattle DARPA-EDCS Workshops, March and July 1997., 1997.

[14] D. L. Métayer. Describing software architecture styles using graph grammars. *IEEE Transactions on Software Engineering*, 24(7):521–553, July 1998.

[15] B. C. Peirce. *Basic Category Theory for Computer Scientists*. The MIT Press, 1991.

[16] J.-C. Raoult and F. Voisin. Set-theoretic graph rewriting. In H.-J. Schneider and H. Ehrig, editors, *Graph Transformations in Computer Science*, pages 312–325. Springer LNCS 776, 1994.

[17] G.-C. Roman, P. J. McCann, and J. Y. Plun. Mobile UNITY: Reasoning and specification in mobile computing. *ACM TOSEM*, 6(3):250–282, July 1997.

[18] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[19] M. Wermelinger and J. L. Fiadeiro. Connectors for mobile programs. *IEEE Transactions on Software Engineering*, 24(5):331–341, May 1998.

[20] M. Wermelinger and J. L. Fiadeiro. Towards an algebra of architectural connectors: a case study on synchronization for mobility. In *Proceedings of the Ninth International Workshop on Software Specification and Design*, pages 135–142. IEEE Computer Society Press, 1998.